NODEJS

Introduction to Node.js

Node.js is a powerful and flexible JavaScript interpreter, capable of letting us run JavaScript code free from our browser, using our computer's native hardware. Using **Chrome's V8 engine**, we can do everything from running JavaScript files directly from the terminal, to traversing local files, to even **spinning up a web server!**

Let me Google that for you

A note before we get started:

As you progress through the platform, you'll be seeing snippets of code that will help you complete the assignments. However, you won't get everything you need on the platform, and **that's intentional**. A successful web developer must be able to solve problems with *incomplete information*. For early assignments, we'll give you some hints as to what you might be missing. For the later challenges you'll have to figure these unknown unknowns out for yourself. Embrace this – it's integral to the art of programming.

Uses

How

As previously mentioned, Node.js uses Chrome's V8 engine to efficiently interpret JavaScript. V8, however, is written in C++ and JS. Why is this important? Well, it allows a relatively high-level language like JavaScript to be compiled to code that is run directly by the CPU! By using this engine, we can skip a bunch of steps in the runtime processes. That means *speed*, but it also means that our JavaScript interpreting scales to the power of our computer!

Sockets

One cool thing about Node.js is its ability to use WebSockets, a technology that allows a continuous, non-blocking connection between the client and server.

If the traditional client-server model is something like morse code, where only one party can send data at a time, non-blocking connections allow both parties to send information at the same time. A phone call is a prime example of non-blocking communication: You can talk to your friend while your friend is speaking to you concurrently. With Node.js, we can easily set up these socket connections and have a persistent connection from each client to a server, which means we can actually force information onto the browser without the need of an HTTP request!

Use cases

Sockets make Node.js a great tool for making real-time applications. These are applications that exchange information as the users input it, with no lag. Things like chat rooms and multiplayer games are great examples of Node's strengths. Also, Node servers are able to support lots of connections. A single node server can hold 200,000 connections whereas a single Apache server tops out at around 20,000 connections.

Drawbacks

Heavy computation is a killer in a Node server, primarily due to poor memory management. In addition, if logic is required before entering the event loop, Node's performance will decline dramatically. Similarly, Node is often used with noSQL databases such as MongoDB, which we'll be using in this course. Mongo also uses a V8 engine core, but it doesn't have the same event-loop (it's just a worker), so large numbers of inefficient queries can back up that worker.
This *could* take up enough memory that it prevents the event-loop from resolving events, making Node much less awesome.

Node also has a very 'batteries-not-included' unopinionated style, favoring configuration over convention. Out of the box, we won't be able to do all the amazing things a full featured framework like Ruby on Rails can do. We'll need to bring in middleware and do our own configuration to really get the most out of Node!

The upshot? You should use Node for what it's good for: making servers capable of handling lots of connections and moving data quickly!

Installation

Before we jump into the nuances of JavaScript, we're going to install Node. Doing this now will give you a couple of new powers as you code the early assignments:

1. You can enter a JavaScript console without needing to open a browser.
2. You can run JavaScript files in your terminal. That means you don't have to write JavaScript in HTML `<script>` tags, which makes the development experience a little cleaner.

Installing software can be cumbersome, but the Node.js install should be pretty straightforward. If you are stuck for some reason, talk to a classmate who has the same type of OS as you.

First, visit http://nodejs.org/ and click install. Once you've opened the file and completed the installation process (which should guide you through it), let's check out our awesome new tool:

Using your terminal, try the following commands after your prompt (`$`):

```
$ which node (where node  for PC users)
```

That should output a path like: `/usr/local/bin/node or \Program Files\nodejs\node.exe`, which tells you where the node is installed.

```
$ whoami
```

That should output your username.

```
$ node -v
```

This should print your current version, which should be at either 6.x LTS (long-term stable) or `v8.x current dev`.

Let's actually enter our own JavaScript environment and declare variables from the comfort of our own terminal.

```
$ node
```

You should have noticed your command prompt may have changed from `$` to `>`. That means whatever we write will be interpreted as JavaScript. (To exit this environment you can just type `ctrl C` twice or `.exit` once.)

Let's write some code:

```
> console.log('hello');
```

Did you see the following output?

```
hello
undefined
```

Wahoo!

One additional note before we move ahead. We just typed `node` to enter a JavaScript environment, but what if we wanted the node to run a file full of code that we wrote in our text editor (which is a bit more efficient than writing directly in the console)? Just append your file name to the `node` command, like so: `node your_file_name.js`.

For macs: If the above commands didn't work (especially `which node`), it's probably your Mac trying to protect you. Use the following command to give yourself the ability to write files into `usr/local/`:

```
sudo chown -R $(whoami) /usr/local/
```

This will prompt you for your password. After you succeed, try reinstalling node!

To exit from node console, press *Ctr+C* or type *.exit* .

NPM

While Node is just a JavaScript Interpreter, it is Node Package Manager, or NPM, that makes it such a powerful development platform. Learning NPM will supercharge our Node by allowing us to bring in new tools.

What is it?

Npm is a package manager and the default tool that comes with Node.js to manage your project dependencies.

What are dependencies? In this case, they're just JavaScript files and libraries that give us tools to make applications with, such as a ready-made function that spins up a server with ease! **That means that npm is just a tool to fetch and prepare other chunks of code**. In the MEAN stack, we call those chunks "**modules**". Depending on what technologies you've used in the past, these "**modules**" are very similar to Ruby **gems** and Python **libraries**, often generalized as "**middleware**".

npm can install and use modules from either a local destination on your computer or it can get them from a remote location called the [npm registry](), an online home for node modules. There are *thousands* of NPM packages that the node community has generated. They can be found here: https://www.npmjs.com/

Installing Packages

To demo installing middleware with npm, we'll use a super useful package called `nodemon`. Using `nodemon` instead of the `node` command in your terminal will automatically *re-run your JavaScript file or project whenever you save something*. That means no more manual server restarts!

This is a node module we will want to be available *everywhere* because we will use it on every project we create. To install nodemon globally, simply type into any command line:

```
$ npm install -g nodemon //(may require sudo)
```

The `-g` is what is called an option. An option is an additional specification we can use on terminal command to refine how it runs. You have seen options before with git:

```
git commit -m  //the -m option is the message option!
```

The option we used is called the 'global' option. The global option run with the `npm install` command and installs the node module in question onto our machine where it can be used anywhere. **Most of the node modules we will install will NOT be global**, if you're not *absolutely* sure if you want it in every Node project you create, *don't add it globally*!

Test nodemon

Using your terminal, make a file called `test.js` and add the following line of code:

```
console.log('I am running from node');
```

Run it with `nodemon` from terminal!

```
$ nodemon test.js
```

Make some changes to `test.js` and save them. What happened in your terminal?

End `nodemon` using `ctrl-C`

Just another reminder: With `node` and `nodemon` we have the power to run JavaScript without having to include it in `<script>` tags of HTML. **NOTE: `node` and `nodemon` script will soon be called server-side script**

Front End

For now manage front-end dependencies utilizing the necessary CDN for the desired library, adding the provided `script` or `link` element to your HTML `head`.

Here are a couple popular libraries:

- Bootstrap: https://www.bootstrapcdn.com/
- jQuery: https://code.jquery.com/

FS and HTTP

What is the FS module?

An essential part of any server is the ability to **read and write files**. Reading a file is how we **obtain** the content to serve to clients, and writing it is how we **output** content to the client. If we don't have a way of doing this, we're not going to be able to build a server! That is why the creators of Node.js built the **fs** (**file system**) module. The FS module allows us to do exactly what we need: **read and write content from files,** and it is by default included in Node.js. It is very rare that you will see the **HTTP** module used without the **fs** module. **The HTTP module is the module that allows us to build a web server that accepts HTTP requests and can serve responses**. Combining the **fs** and **http** modules, we can create simple web servers quite easily.

Setting up a basic server:

Let's make a new folder called node_server and in it make a file called app.js. Here's what goes in app.js:

```
// get the http module:
var http = require('http');
```

```
// fs module allows us to read and write content for responses!!
var fs = require('fs');
// creating a server using http module:
var server = http.createServer(function (request, response){
    // see what URL the clients are requesting:
    console.log('client request URL: ', request.url);
    // this is how we do routing:
    if(request.url === '/') {
        fs.readFile('index.html', 'utf8', function (errors, contents){
            response.writeHead(200, {'Content-Type': 'text/html'});  // send data about respon
se
            response.write(contents);  //  send response body
            response.end(); // finished!
        });
    }
    // request didn't match anything:
    else {
        response.writeHead(404);
        response.end('File not found!!!');
    }
});
// tell your server which port to run on
server.listen(6789);
// print to terminal window
console.log("Running in localhost at port 6789");
```

Great! Now let's make a file called **index.html** and just add some basic content to it. Boot up your node server by navigating to your node_server folder in a terminal window and typing:

```
nodemon app.js
```

This will boot your node server and restart it automatically for you anytime you make changes. When you go to **"localhost:6789"** on your browser, you should see your HTML page loaded. Let's walk through what happened:

THIS LINE IS CRUCIAL:

```
var server = http.createServer(function (request, response){...}
```

This one line creates our web server. This is extremely powerful and concise. You'll notice the **createServer()** method takes a parameter, **namely, a callback function with a request and response parameter**. Hmm...what do you think the request and response parameters are? They are the **HTTP request made by the client and captured by the server and the HTTP response we will prepare and serve back to the client!** So let's sketch out what's going to happen:

1. Any request made to this web server gets passed into the **callback**.
2. **If the request matches one of the response patterns we built into the server**, we will prepare and serve the associated response.
3. If the request doesn't match, we will send back an error to the client.

This pattern is basically the way any web server functions; it has a list of rules to follow regarding the incoming requests and it serves responses according to those rules. Let's build our default request/response pattern. The default, or **root route**, is just the response we serve if we request the basic route of the site. For

instance, **"www.facebook.com"** is the root route for facebook, **"www.facebook.com/trey.villafane"** is **NOT** a root route.

Important: "www.facebook.com" is the same route as "www.facebook.com/"

Configuring the root route:

```
if(request.url === '/') {
    fs.readFile('index.html', 'utf8', function (errors, contents){
        response.writeHead(200, {'Content-Type': 'text/html'});
        response.write(contents);
        response.end();
    });
}
```

The code above captures **ONE REQUEST** and serves **ONE RESPONSE**. Notice the code begins with an **if** statement. We are asking if the **URL property** of the request object is equal to **"/"**. The route "/" **is always considered the root route**. In English, **we are asking if the URL requested by the client is of a particular form**. If the request URL matches the string to the right of the triple equals sign, we will serve the appropriate response. That response begins at fs.readFile(...). **This command goes and finds a file by name and reads it.** The name of the file we're opening is called **'index.html'**. The second parameter is the **encoding** of the file. **Here, we're telling the fs object what type of characters to expect in the file it's opening. You will need to include this line for any text-based document you serve, remember this!!**

When the fs module reads the file, **it passes the contents of the file into a callback and this is where we actually formulate and serve the response**. Notice the first thing we do is call the **response.writeHead()** method. This method sends the **headers** for our response along with a **status code**. **A header is the part of a response that contains the specifics of the response**. We need to tell the browser what type of response we're serving. The status code is a code that tells the browser the status of the response. Any status code in the **200's or 300's is good**. Anything in the **400's to the 500's is bad**. **For now, just always put a 200 as your status code on any valid request.**

After all of that, we finally send the response to the client using the **response.write()** method, which just **sends the contents of the files to the client**. Since a response might contain multiple chunks of data, **we call response.end() when we are finished**. And there you have it! A complete request and response cycle hand-written in Node.js! Awesome, right? This is a lot of code for something we haven't really had to do for ourselves up to this point, but if you want to be a **true full-stack engineer**, you have to know how this stuff is constructed under the hood.

The **server.listen()** method tells our server to listen on a specific port. Once that is configured we are good to go! Now we are serving files!

Node Server

Learning Objectives

- What is a Node server
- How to launch a Node server
- How to respond to requests

Making a Full Web Server

With the information in the previous tab, we were able to create a simple web server that was able to process our request for the root URL of our mini-project. In order to make a fully-fledged web server, you need to write more rules! This can be as simple as adding more else if statements to your code. For example, let's say we wanted to load an HTML file when a client requested the URL: localhost:6789/dojo.html. We could just modify our original code in **app.js** to look like this:

```
// get the http module:
var http = require('http');
// fs module allows us to read and write content for responses!!
var fs = require('fs');
// creating a server using http module:
var server = http.createServer(function (request, response){
    // see what URL the clients are requesting:
    console.log('client request URL: ', request.url);
    // this is how we do routing:
    if(request.url === '/') {
        fs.readFile('index.html', 'utf8', function (errors, contents){
            response.writeHead(200, {'Content-Type': 'text/html'});
            response.write(contents);
            response.end();
        });
    }
    else if (request.url === "/dojo.html") {
         fs.readFile('dojo.html', 'utf8', function (errors, contents){
             response.writeHead(200, {'Content-type': 'text/html'});
             response.write(contents);
             response.end();
         });
    }
    // request didn't match anything:
    else {
        response.end('File not found!!!');
    }
});
// tell your server which port to run on
server.listen(6789);
// print to terminal window
console.log("Running in localhost at port 6789");
```

So now you should be able to make any URL serve any HTML page! But let's say we wanted to serve some **CSS**. What would have to change? Try adding a link tag to your **index.html** page that looks like this:

```
<link rel="stylesheet" type="text/css" href="/stylesheets/style.css">
```

Then run your Node server and bring up your browser and your **JavaScript console**. Open the **network tab** and **refresh your page**. You should see a request made for "**/stylesheets/style.css**". Since our server doesn't have

a rule for that particular URL we're going to see our **"File not found!!!"** message - **this message will be visible if you click on the style.css request and furthermore click on the preview or response tab that opens when you click on the style.css request**. What is important to note here is that a request for a stylesheet is handled exactly the same way as a request for an HTML page. Even though the end result of requesting an HTML page compared to a CSS page is different, **the process is still the same: the server gets a request and sends a response, period**. Use the information in this tab and the previous ones to build out the rest of this server so it can handle this request!

Some hints:

- If you're using jQuery, or Twitter Bootstrap or **anything stored somewhere other than your computer**, you **don't** need to write a route for it on your server. The routes we write in our servers are only for content stored on our servers. Remote stuff is someone else's server's responsibility. Yay!
- **Any file written in plain text will be served with utf-8 encoding**. **Images won't be served with utf-8; omit that argument in the fs.readFile() method when serving images.**
- Use the following table to figure out which headers to send with your server's responses:

File type:	**Headers:**
HTML	{'Content-Type': 'text/html'}
CSS	{'Content-Type': 'text/css'}
Javascript	{'Content-Type': 'text/javascript'}
png/jpeg/gif	{'Content-Type': 'image/*'} (* is the image format, ie png or jpeg etc)

If you feel like going the extra mile, you can read about all the different status codes here. You'll love status code 418.

Assignment: Landing Page

Create a small node server capable of handling the following **request URLs:**

- **localhost:6789/** This route should serve a view file called index.html and display a greeting.
- **localhost:6789/ninjas** This route should serve a view file called ninjas.html and display information about ninjas.
- **localhost:6789/dojos/new** This route should serve a view file called dojos.html and have a form (don't worry about where the form should be sent to).

If the URL is anything other than the ones above, have an error page load saying that the URL requested is not available.

Assignment: Cars and Cats

We are going to make another node server in this assignment, but we are going to **complicate** it. Create a folder called **cars_and_cats**, this will be your **root folder** for this project. Within this folder, create:

- A file called app.js. This is where you will build your **node server**.
- A folder called views. This is where you will keep your **HTML files**.
- A folder called **images** for images.
- A folder called **stylesheets** for CSS.

Your server must be capable of handling the following URL requests:

1. Have **localhost:7077/cars** go to a simple HTML page that shows some cool pictures of different cars. These car pictures should be stored in your **images** folder on your server. **DON'T just link to pictures of cars stored somewhere else!!**
2. Have **localhost:7077/cats** show a simple HTML page with some cool pictures of cats. Again, make sure these pictures are **stored on your server**.
3. Have **localhost:7077/cars/new** show a simple form where the user can add a new car information. For the /cars/new HTML page, no need to store this information in the database nor is there a need to validate the entries. Simply have the form there.

The point of this exercise is to build your node server so it serves all of the content of your website.

Helpful hints

Serving CSS stylesheets

Here is an example of how to handle a browser request for a stylesheet:

```
  else if(request.url === '/stylesheets/style.css'){
    fs.readFile('./stylesheets/style.css', 'utf8', function(errors, contents){
     response.writeHead(200, {'Content-type': 'text/css'});
     response.write(contents);
     response.end();
    })
  }
```

Serving images

```
  else if(request.url === '/images/pizza.jpg'){
    // notice we won't include the utf8 encoding
    fs.readFile('./images/pizza.jpg'
copy
, function(errors, contents){
        response.writeHead(200, {'Content-type': 'image/jpg'});
        response.write(contents);
        response.end();
    })
  }
```

Node Modules

The ability to include code from other files within another is extremely important in a back-end environment. If you recall, we do this with front-end JavaScript by adding `script` tags with the `src` attribute pointing to the right place. But in Node.js, we need to be able to pull code from JavaScript files *into other* JavaScript files.

To do this, we use the `require()` method. This raises the question:

What gets returned when we `require` a file?

Exporting in Node

Let's make a folder called `node_module_basics` and within that folder make two files: `my_module.js` and `app.js`. We are going to import `my_module.js` into `app.js`. Here's our basic set up:

`app.js`
```
var my_module = require('./my_module');
my_module.greet();
```

`my_module.js`
```
module.exports = {
    greet: function(){
        console.log("we are now using a module!");
    }
}
```

Now let's run our Node code by navigating to our `node_module_basics` folder and executing the terminal command `node app.js`.

Things should start clicking now! Your terminal window should show the output "**we are now using a module!**" Let's do some reverse engineering: by **requiring the module my_module, we were able to import the object we set equal to module.exports in the my_module.js file**. This is literally the **most important facet of Node.js;** without this, the **MEAN** stack would be impossible to create. In a sentence:

Requiring a Node module allows you to use the module.exports object of another file!

Let's spice things up a bit. Change your **my_module.js** file to look like this:

```
module.exports = {
    greet: function(){
        console.log("we are now using a module!");
    },
    add: function(num1, num2){
        console.log("the sum is:", num1 + num2);
    }
}
<div id="copy-toolbar-container" style="cursor: pointer; position: absolute; top: 5px; right: 5px; padding: 0px 3px; background: rgba(224, 224, 224, 0.2); box-shadow: rgba(0, 0, 0, 0.2) 0px 2px 0px 0px; color: rgb(187, 187, 187); border-radius: 0.5em; font-size: 0.8em;"><span id="copy-toolbar">copy</span></div>
```

Now make **app.js** look like this:

```
var my_module = require('./my_module');
my_module.greet();
my_module.add(5,6);
```

Run your code with the same command as before. If you don't see a message saying: **"the sum is 11"**, you are in a parallel universe. Clever jokes aside, this should make sense now.

A couple of notes:

You'll notice that we require() the string **"./my_module"**. Two things to note:

1. There is no .js at the end of the file. The require method automatically looks for JavaScript files, so we don't need to include a file extension.
2. The files **app.js** and **my_module.js** are in the same directory. Normally, **the require()** method looks for node modules that aren't in the same directory as the file that is running; by default, the **require() method looks for the modules located in a folder called node_modules.** To tell **require() to look in the current directory** (i.e. the folder that the file is located in) **we have to include "./" in front of the file path. "./" (dot-slash) is the file path for the current directory**. You know how:

```
3. cd ..
```

goes back one directory in terminal navigation? "." is just the current folder. Makes sense, huh?

One more thing

Sometimes, we don't want to export just an object. What if we wanted to return a function that returns an object? Let's switch up our code a little bit:

my_module.js

```
module.exports = function(){
    return {
        greet: function(){
            console.log("we are now using a module!");
        },
        add: function(num1, num2){
            console.log("the sum is:", num1 + num2);
        }
    }
}
```

app.js

```
var my_module = require('./my_module')();      //notice the extra invocation parentheses!
console.log(my_module);
my_module.greet();
my_module.add(5,6);
```

This is another pattern you'll see with requiring and exporting in Node. Now, instead of exporting an **object literal**, we are exporting a **function that returns an object (or an 'object constructor')**. There are situations where one pattern or the other is preferred, but we won't go into that now. **All this process really highlights is the different ways to**

construct JavaScript objects, which you have already had a lot of exposure to! It's like we knew what was important to know when we covered JavaScript!

Other Useful Node Modules

IMPORTANT: *Please spend no more than 30 minutes* going over some of these modules below. Most of these modules you will want to use with frameworks like Express so it's not critical that you know how to use these without using a framework first. The reasons we want you to spend some time learning these modules are **1)** to know that these modules could be used without using any framework like Express and **2)** to know that there are LOTS and LOTS of modules available for you to use (which makes it nice in some ways but also could make things confusing as there can be a lot of different ways to do the same thing).

- **querystring -** this module provides utilities for dealing with query strings (part of the URL that contains the data to be passed) - http://nodejs.org/api/querystring.html
- **url -** this module has utilities for URL resolution and parsing, such as getting the current URL of your page etc - http://nodejs.org/api/url.html
- **redis** - allows you to use a Redis server: https://github.com/mranney/node_redis
- **underscore** - popular utility library: https://npmjs.org/package/underscore
- **coffeescript** - Coffeescript compiler: http://coffeescript.org/
- **node-validator -** is a library of string validation, filtering and sanitation methods.
  Visit https://github.com/chriso/node-validator/blob/master/README.md
- **express** - a nice framework to help you organize your code even further (we'll study this in the next chapter)
- See other node modules you could use: https://npmjs.org/

Assignment: Math Module

For this exercise, create a javascript file called **'mathlib.js'**. We're **not** going to make a node server for this exercise, but we'll still make a file called app.js. Within app.js, we're going to **require our mathlib module** and use its functionality to do some basic computations.

In this javascript module, you're going to return a javascript object that allows the developer to do the following tricks.

- add two numbers (e.g. math.add(2,3) should return 5)
- multiply two numbers (e.g. math.multiply(3,5) should return 15)
- square a number (e.g. math.square(5) should return 25)
- return a random number between the two numbers (e.g. math.random(1,35) should return a random number between 1 and 35)

Now there are a few different ways to do this, but the way we would recommend you structure your 'mathlib.js' is to do something like below:

```
module.exports = function (){
  return {
    add: function(num1, num2) {
         // add code here
    },
    multiply: function(num1, num2) {
         // add code here
    },
    square: function(num) {
         // add code here
    },
    random: function(num1, num2) {
         // add code here
    }
  }
};
```

Math Module in ES6

Now take the previous math module and use ES6 (using Classes) to make it work in mathlib.js.

Note that you may need to update your app.js in how you load/require mathlib.js.

Content Module

As your apps get larger and your server is responsible for serving more files, your app.js will get more complex and harder to read. Instead of having so many if/else statements to serve different static file contents, we are going to create a module that will serve static contents automatically. Later, this type of task is automatically handled by a framework (which you'll learn in the next chapter). Spend up to 4 hours trying to finish this assignment and see how you can build your own module to do these things.

Without your custom module, your app.js would look like below:

app.js

```
//http server
const http = require('http');
const fs   = require('fs');
//creating a server
server = http.createServer(function (request, response) {
  response.writeHead(200, {'Content-type': 'text/html'});
  console.log('Request', request.url);
  if(request.url === '/'){
    fs.readFile('views/index.html', 'utf8', function (errors, contents) {
      response.write(contents);
      response.end();
    });
  } else if(request.url === '/dojo.html'){
    fs.readFile('views/dojo.html', 'utf8', function (errors, contents) {
      response.write(contents);
```

```
      response.end();
    });
  } else if(request.url === '/stylesheet/style.css'){
    fs.readFile('stylesheet/style.css', 'utf8', function (errors, contents) {
      response.write(contents);
      response.end();
    });
  } else {
      response.end('File not found!!!');
  }
});
server.listen(8000);
console.log("Running in localhost at port 8000");
```

After your change (or after you create your static.js in the modules folder), your app.js would look like the following:

```
// http server
const http = require('http');
const   fs = require('fs');
// the file below is the file you need to create for this assignment.
// NOTE!!!  The '.' in the filepath below just refers to the location of the current file, in
this case
// the file is app.js.  Thus the path './static.js' just refers to a file called static.js
const static_contents = require('./static.js');
//
//creating a server
server = http.createServer(function (request, response){
  static_contents(request, response);  //this will serve all static files automatically
});
server.listen(8000);
console.log("Running in localhost at port 8000");
```

Make your static.js file as elegant as possible. We think with proper refactoring you could make this work for hundreds/thousands of files with just a few lines of code.

IMPORTANT - Your code should work when we add new files in the views folders, or if we were to add new images (e.g. /images/car.jpg), new stylesheets (e.g. /stylesheets/others.css), etc. If the file the user is seeking is not there, it should also render an error message.

The goal of this assignment is for you to create your own first custom node module and for you to not rely so much on all the node modules out there (as it's just a library like this that other people wrote) but know how to create these node modules yourself if you wanted to. Creating your own node modules would make your app.js simpler but also allow your app to be more modularized. It's always better as the app is getting complicated to have your own codes do the things you need, rather than relying on 3rd party libraries someone else wrote (especially if their documentations are really poor).